Bertrand Saintes
Master 1 Informatique

ORDONNANCEMENT DE TACHES

-

Mise en oeuvre d'un algorithme d'ordonnancement efficace résistant aux pannes dans un réseau hétérogène

Tuteurs : M. Nakechbandi, J.-Y. Colin

1. Introduction

La vie de tous les jours regorge d'une multitude de situations nécessitant l'allocation de ressources à un certain nombre de tâches. La gestion de projet en est le plus criant exemple, aussi bien humain que matériel, voire les deux dans la majorité des cas et ce quel que soit le domaine. Ainsi, la gestion de projet peut concerner aussi bien le bâtiment que l'informatique ou la médecine. Ce problème se pose à chaque seconde à l'intérieur d'un ordinateur où le système d'exploitation doit en permanence allouer du temps processeur aux différentes tâches afin qu'elles mènent toutes à bien leurs calculs.

Or, ce type de problèmes impose plusieurs contraintes dont la plus évidente de toutes : la précédence. Cette contrainte impose que toutes les tâches *k* qui précèdent une tâche *i* donnée doivent avoir terminé leur exécution avant de pouvoir lancer l'exécution de *i* afin qu'elle dispose de tous leurs résultats. Plusieurs algorithmes permettent de modéliser et de résoudre ce type de problème. Toutefois, qu'advient-il de la solution lorsqu'une machine tombe en panne ? Comment être certain d'obtenir le résultat final si la chaîne d'exécution risque d'être interrompue ?

Afin de répondre à cette interrogation, M. Nakechbandi, J.-Y. Colin et J.B. Gashumba proposent d'adapter un algorithme intitulé **eFRCD** mis au point par X. Qin et H. Jiang à leur algorithme d'ordonnancement optimal **DSS_OPT**, algorithmes que nous étudierons plus bas.

Ainsi, nous verrons en premier lieu les objectifs et enjeux liés à la mise en place d'un algorithme d'ordonnancement résistant aux pannes. Puis nous verrons les différentes hypothèses à considérer pour mettre en oeuvre ces algorithmes. Nous verrons ensuite comment a été réalisé l'implémentation de ces derniers. Enfin nous aborderons les différents critiques et observations ainsi que les évolutions possibles en découlant.

2. Objectifs

La mise en place de telles solutions répond à plusieurs objectifs. Tout d'abord, la première question qui vient à l'esprit est : *comment utiliser de la meilleure façon qui soit les ressources disponibles afin d'exécuter toutes les tâches dans un cadre hétérogène* ? De cette question en découle immédiatement une seconde : *comment planifier l'exécution de toutes les tâches en respectant la condition de précédence* ?

On voit donc apparaître deux objectifs principaux, à savoir la meilleure répartition possible des ressources affectées aux tâches ainsi que la meilleure planification possible des tâches tout cela dans le but d'obtenir le résultat final le plus tôt possible.

De plus, une autre question se pose lorsque l'on observe ce type de problème dans la réalité : *comment être sûr que, en cas de panne d'une ressource, on obtienne bien un résultat final* ? Ce problème se pose par exemple lorsque l'on effectue une suite de calculs ou d'opérations distribués sur différents serveurs répartis dans le monde. En effet, une panne peut tout à fait survenir sur un des serveurs provoquant alors une faille dans l'ordonnancement car la tâche affectée sur cette machine ne s'exécutera pas et n'enverra donc pas ses résultats soit à l'utilisateur dans le cas d'une tâche finale, soit à son ou ses successeurs. Ainsi, il faudrait être sûr que l'ordonnancement puisse prendre en compte ce type de situations. Ici, l'objectif n'est plus d'obtenir une solution optimale résistante aux pannes mais simplement être sûr d'obtenir le résultat final quoi qu'il en coûte.

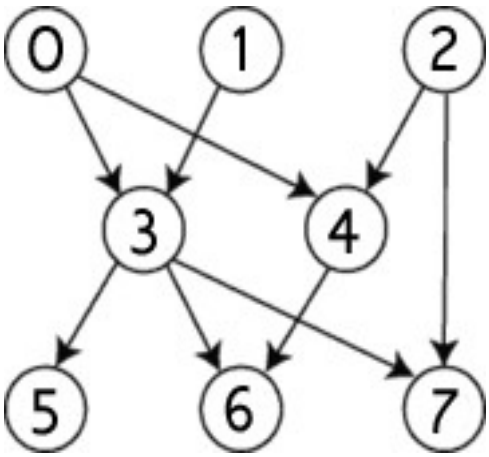
3. Hypothèses de base

Afin d'implémenter correctement les deux algorithmes qui nous intéressent ici, plusieurs hypothèses sont à formuler. Pour commencer, le programme complet doit être divisé en plusieurs tâches dont certaines dépendent d'autres tâches. Tout d'abord, le système que l'on souhaite modéliser est un réseau hétérogène. Cela signifie que l'exécution d'une tâche donnée peut être plus ou moins lente selon la machine sur laquelle elle est exécutée. En effet, différents types de machines peuvent être disponibles, certaines disposant d'un câblage spécifique permettant d'exécuter un type d'opération beaucoup plus rapidement qu'un autre par exemple. On obtient donc des temps d'exécution par tâche très différents selon les machines. Il est également envisageable qu'une tâche puisse ne pas être exécutable sur tous les serveurs. Il faut néanmoins qu'au moins deux d'entre eux soient capables de l'exécuter comme nous le verrons plus bas.

De plus, on considère que le réseau sur lequel on distribue nos tâches est suffisamment étendu pour que le temps de communication d'une unité de données entre deux machines données ne soit pas négligeable. Ainsi, deux machines éloignées mettront plus de temps pour communiquer entre elles que deux machines situées dans le même réseau local. Il faut également ajouter que le temps de communication entre deux machines est le même quel que soit le sens dans lequel l'information circule. Cela nous permet de modéliser un réseau proche des conditions constatées sur un réseau étendu. Enfin, on admet que la communication de deux tâches dépendantes l'une de l'autre et s'exécutant sur la même machine est tellement négligeable qu'on le considère comme nul.

Ensuite, le système modélisé ne dispose pas de contraintes de ressources ce qui signifie que plusieurs tâches peuvent s'exécuter au même instant sur un même serveur. En effet, on constate dans les faits que les serveurs disponibles sur Internet sont capables d'exécuter plusieurs centaines de requêtes au même instant (machines multi-processeurs par exemple). Ainsi, ajouter une exécution supplémentaire ne pose pas de problèmes pour ce type de machines. On considère donc que deux tâches exécutées sur un même serveur n'entrent pas en concurrence.

Afin de modéliser la relation de précédence entre les différentes tâches, nous avons recours à un graphe acyclique orienté (Directed Acyclic Graph). Il s'agit d'un graphe ne possédant pas de cycles et donc les arcs sont orientés comme le montre l'illustration ci-dessous :



Graphe acyclique orienté (DAG)

De plus, on s'assure que le graphe est connexe. En effet, un graphe non connexe perturberait l'ordonnancement dans la mesure où chaque composante connexe peut être considérée comme un sous-problème indépendant des autres. On s'arrange pour que l'indice d'une tâche donnée soit inférieur à tous ceux de ses successeurs mais également supérieur à ceux de ses prédécesseurs. Cette hypothèse a une grande utilité dans l'implémentation du graphe comme nous le verrons plus tard.

La quantité de données à transmettre entre les tâches est symbolisée dans le cas qui nous intéresse par des poids sur les arcs. Ainsi, le coût réel de communication entre deux tâches k et i s'exécutant respectivement sur deux machines $sigma_p$ et $sigma_r$ est symbolisé par le coût de transfert d'une unité de données entre $sigma_p$ et $sigma_r$ multiplié par la quantité de données que la tâche k envoie à la tâche i. Ainsi, en considérant une des hypothèses citées plus haut, deux tâches s'exécutant sur une même machine vont engendrer un coût de communication réel nul quelque soit la quantité d'informations envoyée de l'une à l'autre.

4. Mise en oeuvre

1. Étape préliminaire

Avant d'implémenter les algorithmes d'ordonnancement, il a fallu modéliser le système qui nous sert de support. Ainsi, plusieurs questions se sont posées telles que comment modéliser le graphe, comment stocker les temps d'exécution des tâches sur chaque machine, le coût de transmission entre deux machines du réseau etc. Pour ce faire, j'ai eu recours à trois matrices principales :

- *la matrice lien[][]* : cette matrice carrée contient le volume de données à transmettre entre deux tâches quelconques du graphe. C'est elle qui symbolise les arcs entre les tâches. Si aucun arc n'existe entre deux tâches quelconques alors le volume de données est nul. On obtient une matrice triangulaire supérieure car par hypothèse, on a précisé que l'indice d'une tâche quelconque est nécessairement supérieur à ses prédécesseurs et nécessairement inférieur à ses successeurs.
- *la matrice comm[][]* : cette matrice carrée stocke le coût de transfert d'une unité de donnée entre deux machines quelconques. Ici, on constate que la diagonale est nulle car, par hypothèse, le coût de transfert d'une unité de donnée d'une machine à elle même est nul.
- *la matrice tache[nombre de tâches][nombre de machines]* : cette matrice non carrée stocke le coût d'exécution de toutes les tâches sur chacune des machines. Si une tâche quelconque ne peut être exécutée par une machine quelconque, on place un *-1* dans la cellule correspondante afin de savoir que la tâche ne peut être exécutée sur cette machine.

Une fois les matrices créées, il est facile d'obtenir les successeurs et prédécesseurs de chacune des tâches, mais également de connaître les machines capables d'exécuter une tâche donnée, etc. Plusieurs méthodes « utilitaires » de ce genre ont été écrites afin de simplifier les actions lors de l'implémentation des algorithmes d'ordonnancement. Vous trouverez, jointe avec le projet, la *Javadoc* du projet contenant une description détaillée pour chacune des méthodes du projet.

De plus, lorsque le programme remplit aléatoirement les matrices, il écrit ces dernières dans un fichier de trace permettant par la suite de le charger afin d'effectuer des tests. Il est également possible d'écrire son propre fichier de trace à condition de respecter le format imposé comme le montre l'exemple suivant :

```
[MAX_TACHES]3
[MAX_PROCS]3
[Matrice de communication :]
0 3 3
3 0 1
3 1 0

[Matrice de taches :]
27 28 16
20 17 13
22 23 20

[Matrice de liens :]
0 2 0
0 0 8
0 0 0
```

On observe donc qu'il faut tout d'abord préciser le nombre de taches puis le nombre de serveurs à la ligne suivante. Enfin, on saisit les trois matrices en respectant les règles suivantes :

- la matrice de communication est carrée;
- la matrice de tâches n'est pas carrée;
- la matrice de liens est carrée;

Il est également important de noter que chaque fin de ligne remplie par l'utilisateur doit se terminer par un chiffre suivi immédiatement d'un saut de ligne, il faut donc prendre garde à ne pas laisser un espace traîner sans quoi une erreur se produira.

2. DSS_OPT

Une fois le système complètement modélisé, il est possible de passer à l'implémentation du premier algorithme d'ordonnancement, à savoir **DSS_OPT**. Vous trouverez en annexe le pseudo-code basique de cet algorithme. Le but de cet algorithme est de fournir une solution optimale à un problème P d'ordonnancement. Pour cela, on distingue deux phases. Tout d'abord, on commence par déterminer les dates d'exécution et de fin d'exécutions au plus tôt de chaque tâche sur chacune des machines en parcourant le graphe de tâche en tâche en commençant par celles qui n'ont pas de prédécesseurs. Ces résultats sont stockées dans deux matrices, l'une contenant les dates d'exécution, la seconde contenant les dates de fin d'exécution.

Une fois cette première partie terminée, on analyse récursivement pour chaque tâche en démarrant par celles qui n'ont pas de successeurs le serveur lui permettant de finir le plus tôt possible et vérifiant que ses prédécesseurs aient bien eu le temps d'envoyer leurs données. Finalement, une fois cette partie terminée, on a la garantie d'obtenir un ordonnancement qui est optimal car l'algorithme aura étudié pour chaque tâche la machine permettant de terminer au plus tôt en respectant la contrainte de précédence. Cet ordonnancement est stocké et retourné sous la forme

d'un vecteur où chaque cellule de ce dernier est elle-même un vecteur contenant les informations utiles, à savoir : l'indice de la tâche, l'indice de la machine sur laquelle elle sera exécutée, la date à laquelle elle est programmée et enfin la date de fin d'exécution.

3. eFRCD

Bien que les résultats expérimentaux fournis par **DSS_OPT** montrent la présence par moments de plusieurs exemplaires d'une même tâche s'exécutant sur différentes machines, on ne peut être sûr qu'en cas de panne sur un des serveurs on obtiendra le résultat souhaité. C'est pour cette raison que M. Nakechbandi, J.-Y. Colin et J.B. Gashumba ont eu l'idée d'adapter un algorithme existant intitulé **eFRCD** développé par X. Qin et H. Jiang. Ce dernier avant-tout utilisé dans le domaine de l'ordonnancement *temps réel* avec de fortes contraintes de temps a pour but de fournir un ordonnancement résistant à une panne survenant sur une des ressources disponibles. C'est à dire que quelque soit la machine qui disparaît, on pourra toujours obtenir un résultat final. L'intérêt ici est de coupler à l'optimalité de **DSS_OPT** la robustesse de l'algorithme **eFRCD**. Ainsi, si aucune panne ne survient, notre ordonnancement est optimal et en cas de panne, nous sommes assurés que le calcul se termine.

Pour cela, l'algorithme **eFRCD** va placer des copies « backups » des tâches ne disposant pas déjà de plus d'une copie placée par **DSS_OPT** dans l'ordonnancement. Une fois la liste des tâches à dupliquer obtenue, il faut trouver quelles seront les machines susceptibles de faire tourner ces backups. Pour cela, l'algorithme repose sur une heuristique permettant d'élire la machine recevant le backup. Il faut noter que l'on interdit l'exécution d'un backup sur la même machine que la tâche dont il est issu afin de garantir la résistance à une panne. On regarde donc, pour chaque tâche, les serveurs autres que celui utilisé pour exécuter cette dernière puis on choisit celui qui permet d'exécuter le plus rapidement possible le backup. Prenons par exemple une tâche i exécutable sur 3 serveurs : $sigma_1$,$sigma_2$,$sigma_3$. Si la copie primaire de i est placée par **DSS_OPT** sur $sigma_1$, alors les deux serveurs restants pour exécuter une copie backup de i sont : $sigma_2$ et $sigma_3$. Ainsi, l'heuristique choisira le serveur pour lequel on minimise le temps d'exécution de i. Une fois la nouvelle machine déterminée pour toutes les tâches, il reste à définir la date à laquelle on exécutera les copies. Pour cela, il faut prendre en compte deux choses, à la fois les prédécesseurs dits « classiques » c'est à dire les copies primaires des prédécesseurs mais également les backups des prédécesseurs. Ainsi, si on reprend la tâche i ayant pour prédécesseurs a et b et i' le backup de i, il faut s'assurer qu'à la fois a et b aient le temps de finir leur exécution et d'envoyer leurs données à i' mais également que a' et b' aient fini leur exécution et envoyé leurs données à i'. On calcule donc les dates auxquelles TOUS les successeurs terminent leur exécution sans oublier le temps de

transfert des données envoyées à *i'*. On garde le maximum de toutes ces dates, ce qui nous donne la date au plus tôt à laquelle démarrer *i'* en étant sûr d'avoir reçu toutes les données de tous les prédécesseurs. Une fois la date déterminée, on ajoute dans un vecteur contenant l'ordonnancement résistant aux pannes le backup avec ses informations. On obtient en définitive un ordonnancement capable de résister à une panne survenant sur n'importe quelle machine du réseau.

4. Sortie XML

Afin de ne pas perdre le résultat ainsi obtenu, il nous est apparu qu'une sauvegarde de l'ordonnancement au format **XML** (Extensible Markup Language) était une bonne idée. En effet, ce langage informatique fonctionne avec des balises génériques que l'on définit soit même. Il est donc facile de stocker n'importe quel type de contenu en balisant correctement son fichier. Dans le cas qui nous intéresse, quelques balises suffisent à encadrer notre ordonnancement :

- *<schedule>...</schedule>* : balise générale englobant toutes les autres, elle possède un attribut name spécifiant le nom de l'ordonnancement;
- *<resources>...</resources>* : balise spécifiant la liste des ressources disponibles pour réaliser les tâches;
- *<resource id="_" name="_"/>* : balise permettant de détailler une ressource en spécifiant son identifiant ainsi que son nom;
- *<tasks>...</tasks>* : balise spécifiant la liste des tâches à exécuter;
- *<task id="_" name="_" start="_" duration="_" resource="_">* **:** balise permettant de détailler chacune des tâches;

Encore en développement lors de la rédaction de ce rapport, la sortie XML n'est pas encore complète. En effet, il reste à ajouter dans le fichier les arcs du graphe de tâches, à savoir les communications entre les différentes tâches afin de permettre à un logiciel générant des diagrammes de Gantt d'utiliser nos résultats pour obtenir un rendu visuel.

5. Exemple numérique

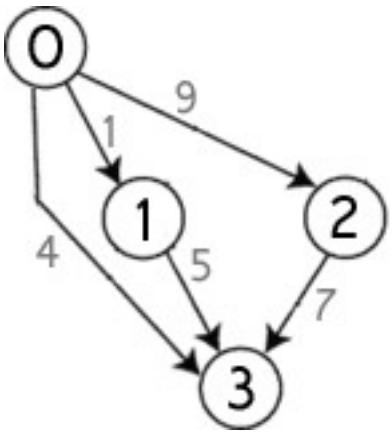
Afin d'illustrer le fonctionnement de ces deux algorithmes, nous allons prendre un exemple numérique simple basé sur l'ordonnancement de quatre tâches avec trois machines à disposition. Voici le fichier de trace tel que le programme l'a écrit une fois les matrices initialisées :

```
[MAX_TACHES]4
[MAX_PROCS]3
[Matrice de communication :]
0 1 3
1 0 1
3 1 0

[Matrice de taches :]
29 27 18
14 27 27
19 10 12
23 13 12

[Matrice de liens :]
0 1 9 4
0 0 0 5
0 0 0 7
0 0 0 0
```

Au vue de la matrice de liens, on peut déduire le graphe acyclique orienté avec la pondération des arcs suivant :



Soit l'ensemble des tâches distribuées $I = \{0,1,2,3\}$ et l'ensemble des machines disponibles $\sum = \{sigma_0, sigma_1, sigma_2\}$ avec les temps d'exécution suivants :

Temps d'exécution $_{i/\sigma sigma\_r}$	***sigma*$_0$**	***sigma*$_1$**	***sigma*$_2$**
0	29	27	18
1	14	27	27
2	19	10	12
3	23	13	12

On obtient également les tableaux suivants indiquant les temps de communication réels:

(0,1)	**sigma$_0$**	**sigma$_1$**	**sigma$_2$**
sigma$_0$	0	1	3
sigma$_1$	1	0	1
sigma$_2$	3	1	0

(0,2)	**σsigma$_0$**	**σsigma$_1$**	**sigma$_2$**
σsigma$_0$	0	9	27
σsigma$_1$	9	0	9
σsigma$_2$	27	9	0

(0,3)	**sigma$_0$**	**sigma$_1$**	**sigma$_2$**
sigma$_0$	0	4	12
sigma$_1$	4	0	4
sigma$_2$	12	4	0

(1,3)	**sigma$_0$**	**sigma$_1$**	**sigma$_2$**
sigma$_0$	0	5	15
sigma$_1$	5	0	5
sigma$_2$	15	5	0

(2,3)	**sigma$_0$**	**sigma$_1$**	**sigma$_2$**
sigma$_0$	0	7	21
sigma$_1$	7	0	7
sigma$_2$	21	7	0

Par exemple, le coût de la communication de la tâche *2* à *3*, s'exécutant respectivement sur sigma$_0$ et sigma$_2$ est égal à 21 car la tâche *2* envoie 7 unités de données et le coût de transfert d'une unité d'information entre sigma$_0$ et sigma$_2$ est de 3. Ainsi, on obtient 7 * 3 = 21 unités de temps.

La première partie de l'algorithme DSS_OPT détermine pour toutes les tâches sur tous les serveurs la date la plus tôt possible d'exécution, il en résulte les deux matrices b et r suivantes :

0	**b0**	**r0**
sigma$_0$	0	29
sigma$_1$	0	27
sigma$_2$	0	18

1	**b1**	**r1**
sigma$_0$	21	35
sigma$_1$	19	46
sigma$_2$	18	45

2	**b2**	**r2**
sigma$_0$	29	48
sigma$_1$	27	37
sigma$_2$	18	30

3	**b3**	**r3**
sigma$_0$	44	67
sigma$_1$	40	53
sigma$_2$	45	57

On recherche ensuite la date de fin d'exécution des tâches sans successeurs la plus tôt possible puis on conserve la date la plus grande. On sait en observant le graphe que la dernière tâche est la *3*, on regarde donc pour chaque machine capable d'exécuter cette tâche, celle qui retourne la date de fin d'exécution la plus tôt. On cherche donc dans r_3 la valeur minimale, à savoir 53. On fixe donc le makespan à 53, ce qui correspond à la date à laquelle on peut terminer le programme au plus tôt.

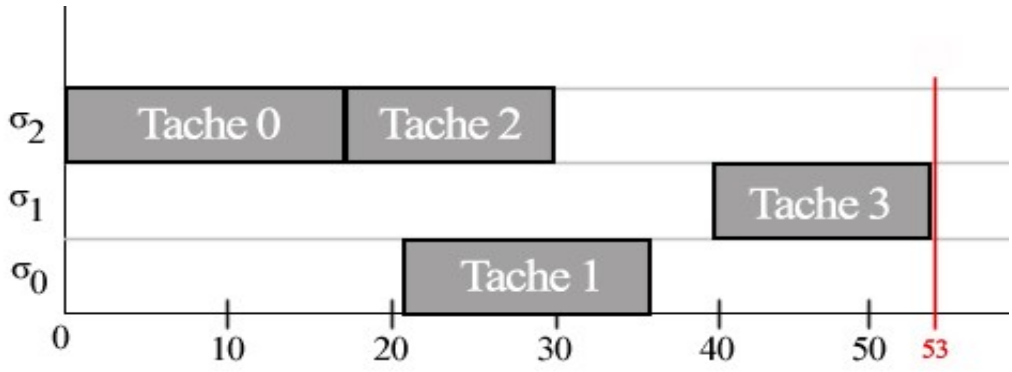
Dans la seconde partie de l'algorithme, on récupère toutes les tâches sans successeurs. Ici, cet ensemble se réduit à la tâche *3* puis on récupère l'ensemble $\sum_3 = \{sigma_0, sigma_1, sigma_2\}$ et on cherche les machines dans cet ensemble capable de faire terminer cette tâche au plus tard à 53. Une seule machine permet cela : $sigma_1$. On programme donc l'exécution de la tâche *3* sur le serveur $sigma_1$ à la date b_3, soit 40.

Ensuite, on constate que les tâches *0, 1* et *2* sont les prédécesseurs de *3*. Pour *0*, les trois machines capables de faire tourner cette tâche permettent de respecter la contrainte de précédence. Avec toutes les machines permettant de respecter la contrainte de précédence on crée une liste, ici on obtient : $\{0/sigma_0, 0/sigma_1, 0/sigma_2\}$. On tire ensuite un élément de cette liste afin de programmer l'exécution de la tâche 0 sur une des machines. Cette notion de tirer un élément dans cette liste peut être modifier afin d'obtenir des résultats différents comme nous le verrons plus tard.

Ensuite, on regarde les machines capables d'exécuter la tâche *1* en respectant la contrainte de précédence, c'est à dire se terminer et envoyer ses données à la tâche *3* avant 40. Seul l'exécution de la tâche *1* sur $sigma_0$ permet cela. En effet, $(r_{1/sigma0} + C_{1/sigma0,\ 3/sigma1}) <= 40$ puisque $r_{1/sigma0}$ vaut 35 et que $C_{1/sigma0,\ 3/sigma1}$ vaut 5. On programme donc l'exécution de la tâche *1* sur $sigma_0$ à la date 21 et ainsi de suite en remontant le graphe. Finalement, on obtient le tableau suivant résumant l'ordonnancement optimal obtenu par l'algorithme DSS_OPT :

	$0/sigma_2$	**$1/sigma_0$**	**$2/sigma_2$**	**$3/sigma_1$**
$b_{i/sigmar}$	0	21	18	40
$r_{i/sigmar}$	18	35	30	53

Ce tableau peut être représenté sous la forme d'un diagramme de Gantt comme le montre l'illustration ci-dessous :



Ordonnancement optimal retourné par DSS_OPT

L'algorithme DSS_OPT permet donc comme on vient de le voir d'obtenir une solution optimale. Maintenant, penchons-nous sur l'application de l'algorithme eFRCD afin de rendre cet ordonnancement résistant à une panne. Tout d'abord, on compte le nombre de copies de chaque tâche placées par DSS_OPT. Ici, on constate que chaque tâche n'est présente qu'une fois dans l'ordonnancement. Étudions d'abord la tâche *0*, la copie primaire placée par DSS_OPT s'exécute sur $sigma_2$, et on constate que $sigma_1$ et $sigma_0$ sont également capables d'exécuter cette tâche, respectivement en un temps de 29 et 27. On va donc élire $sigma_1$ pour exécuter la copie « backup » de la tâche *0*. On répète l'opération pour toutes les tâches qui ne sont présentes qu'une fois dans l'ordonnancement calculé par DSS_OPT. L'ensemble de ces opérations donne le tableau récapitulatif suivant :

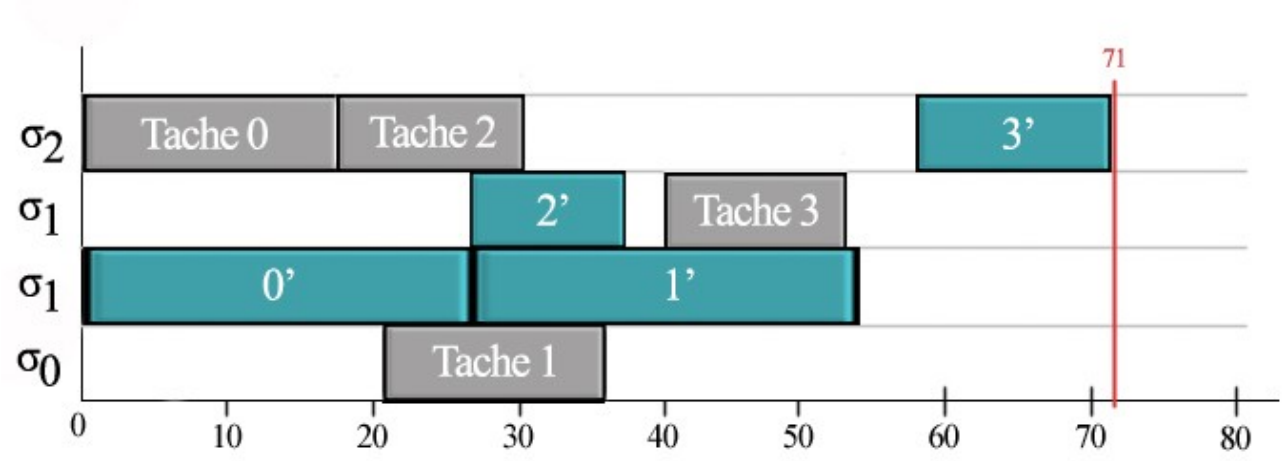
Indice de la tâche	**0**		**1**		**2**		**3**	
Candidats à l'exécution	0	*1*	*1*	2	0	*1*	0	*2*
Coût d'exécution	29	*27*	*27*	27	19	*10*	23	*12*
Élection	*1*		*1*		*1*		*2*	

Maintenant que l'on a trouvé sur quelle machine exécuter chaque backup, il nous reste à déterminer la date à laquelle on programme leur exécution. Pour cela, on démarre à partir des tâches sans prédécesseurs, ici on commence par la tâche *0*. Ne disposant pas de prédécesseurs, on peut programme la tâche *0′* à la date 0 sur $sigma_1$ qui terminera son exécution à la date 27. Ensuite, nous étudions le cas de la tâche *1* qui s'exécute sur $sigma_0$. On cherche donc à placer la tâche *1′* sur $sigma_1$, copie backup de notre tâche *1*. On sait que cette tâche a pour prédécesseurs la tâche *0* mais également la tâche *0′*. Ainsi, on calcule que la tache *0* sur $sigma_2$ aura terminé et envoyé ses données à la tâche *1′* sur $sigma_1$ au plus tôt à la date 18 + 1 soit 19. Ensuite, on constate que la tâche *0′* sur

$sigma_1$ aura fini et envoyé ses données à *1'* au plus tôt à la date 27 + 0 (ici le coût de communication entre *0'* et *1'* est nul puisque, par hypothèse, les deux tâches s'exécutent sur la même machine) soit 27. On dispose donc de l'ensemble suivante : {19,27} dans lequel on cherche la valeur maximale soit 27. A partir de cette date, on est assuré que tous les prédécesseurs de la tâche *1'* aient eu le temps de terminer leur exécution et d'envoyer leurs données. On programme donc l'exécution de la tâche *1'* de la date 27 à 54. On réitère ensuite l'opération pour les tâches *2* et *3*. Finalement, on obtient l'ordonnancement suivant :

	$\mathbf{0}_{sigma2}$	$\mathbf{0'}_{sigma1}$	$\mathbf{1}_{sigma0}$	$\mathbf{1'}_{sigma1}$	$\mathbf{2}_{sigma2}$	$\mathbf{2'}_{sigma1}$	$\mathbf{3}_{sigma1}$	$\mathbf{3'}_{sigma2}$
$\mathbf{b}_{i/sigmar}$	0	0	21	27	18	27	40	59
$\mathbf{r}_{i/sigmar}$	18	27	35	54	30	37	53	71

Cet ordonnancement montre bien que quelque soit le serveur subissant une panne, il est toujours possible de terminer le calcul. On peut représenter cet solution sous la forme d'un nouveau diagramme de Gantt simplifié et complété avec les backups :



Ordonnancement retourné par eFRCD

On constate que le makespan est reculé dans le temps afin de permettre aux backups de terminer leur exécution. Une fois les backups placés, un fichier XML récapitulatif est généré. Le fichier XML correspondant à cet exemple est disponible en annexe.

6. Analyse de résultats

Afin d'obtenir des résultats interprétables, on effectue ici plusieurs exécutions du programme dans différentes conditions. En effet, on répète 20 fois chaque situation puis on calcule la moyenne des makespans obtenus avec DSS_OPT et eFRCD et enfin le temps moyen consommé par l'algorithme comme le résume le tableau ci-dessous :

- Makespan moyen et écart-type obtenus avec DSS_OPT

Machines \ Tâches	**500**		**50**	
50	*3951,75*	*79,28*	*396,75*	*48,15*
5	*4692,1*	*174,28*	*511,90*	*38,53*

- Makespan moyen et écart-type obtenus avec eFRCD

Machines \ Tâches	**500**		**50**	
50	*3986.65*	*79,95*	*435*	*56,56*
5	*4889,85*	*175,94*	*563,85*	*39,99*

- Temps moyen d'occupation CPU et écart-type par DSS_OPT

Machines \ Tâches	**500**		**50**	
50	*12.568 secondes*	*0,31*	*0,07895 secondes*	*0,01*
5	*0,30005 secondes*	*0,01*	*0,00255 secondes*	*0,01*

- Temps moyen d'occupation CPU et écart-type par eFRCD

Machines \ Tâches	**500**		**50**	
50	*0,7273 secondes*	*0,31*	*≈ 0 secondes*	*0,01*
5	*0.02725 secondes*	*0,01*	*≈ 0 secondes*	*0,01*

Plusieurs constatations peuvent être faites au vu de ces tableaux. Tout d'abord, on constate que bien que le placement des tâches backups fait reculer le makespan dans le temps et ce que quelque soit la taille du problème. De plus, on constate également que le nombre de machines inséré dans le problème influe sur le makespan d'une manière générale ce qui contredit une des hypothèses de base, à savoir la non contrainte de ressources. En effet, dans la mesure où l'on peut superposer *n* exécutions au même instant sur une même machine, le makespan ne devrait pas être si sensible à la quantité de machines à disposition...Ce n'est pourtant pas ce que l'on constate avec un makespan moyen de 3986.65 pour 500 tâches sur 50 machines à disposition et 4889,85 pour le même nombre de tâches sur 5 machines. Ceci s'explique peut-être par la manière dont la matrice stockant les coûts de communications entre les machines est remplie. Dans tous les cas, cela ne remet pas en cause le fonctionnement de l'algorithme lui-même qui fournit bien un ordonnancement résistant à une panne.

Des tableaux de résultats précédents, on peut également estimer la rapidité de l'algorithme à fournir à la fois une solution optimale ainsi qu'une solution résistante aux pannes. On constate que l'algorithme demande un certain temps qui est à la fois fonction du nombre de tâches et du nombre de machines, il est d'ailleurs difficile de déterminer dans quelle mesure chacun influence le résultat. De plus, on remarque rapidement que la majorité du temps CPU est consommé pour établir la solution optimale. En effet, si l'on reprend le cas où 500 tâches sont à placer sur 50 machines disponibles, le temps utilisé pour appliquer l'algorithme eFRCD sur la solution retournée par DSS_OPT ne représente que 6% du temps total. On peut dresser le même constat sur les autres situations, ce pourcentage devenant quasiment nul sur des problèmes de plus petites tailles

.

7. Observations et critiques

Plusieurs observations et critiques peuvent être formulées à propos de ces deux algorithmes. Tout d'abord, le système modélisé peut être critiquer. En effet, les auteurs ont posées plusieurs hypothèses avantageuses empêchant l'utilisation de l'algorithme DSS_OPT dans n'importe quelle situation. En effet, poser comme hypothèse que plusieurs exécutions sur une même machine au même instant n'entrent pas en concurrence restreint d'emblée l'application de l'algorithme à des machines particulières telles que les machines multi-processeurs par exemple ou d'autres types de serveurs. Il est donc impossible de l'utiliser pour des projets sollicitant la machine d'un internaute lambda tels que *Folding@Home* par exemple. De plus, la complexité de DSS_OPT en $O(n^2s^2)$ où n représente le nombre de tâches à placer et s le nombre de machines disponibles interdit l'utilisation de cet algorithme sur un problème de taille importante. En effet, on constate dans les faits que l'exécution de l'algorithme DSS_OPT consomme beaucoup de temps CPU afin d'élaborer la solution. Toutefois, en dépit de ces inconvénients, on a la garantie d'obtenir une solution optimale. De plus, une amélioration a été apportée par rapport au pseudo-code de base lorsque dans la seconde partie, on programme les tâches de manière récursive. En effet, prenons une tâche *i* quelconque d'un graphe de tâches. Une fois l'exécution de *i* programmée sur $sigma_r$, on récupère tous les prédécesseurs de *i* notés comme suit : PRED(*i*). Puis, pour chaque *k* appartenant à PRED(*i*), on récupère les machines capables d'exécuter *k* que l'on note : $\sum_k$. Ensuite, pour chaque $sigma_p$ appartenant à $\sum_k$ on stocke dans une liste le couple $\{k/sigma_p\}$ si la contrainte de précédence avec $\{i/sigma_r\}$ est respectée. Une fois tous les prédécesseurs vérifiés, le pseudo-code indique que l'on tire un élément d'une manière quelconque dans cette liste. La modification apportée ici consiste à tirer uniquement le meilleur élément de cette liste, à savoir le couple $\{k/sigma_p\}$ permettant de finir le plus tôt possible en respectant la contrainte de précédence. Ainsi, il n'est plus nécessaire de construire la liste au fur et à mesure que l'on examine les *k* successifs puis de tirer un élément, mais simplement de stocker le couple permettant d'obtenir la date permettant de terminer le plus tôt possible lorsque l'on étudie les prédécesseurs de *i*. Cela permet de s'affranchir de l'utilisation d'une structure pour stocker la liste. Plusieurs méthodes peuvent probablement être envisagées pour tirer l'élément dans cette liste. On peut par exemple tirer un élément de manière aléatoire, ou prendre le premier si on souhaite utiliser une pile, ou encore prendre le dernier si on utilise une file par exemple.

L'implémentation d'eFRCD a pour avantage de générer un ordonnancement résistant à une panne. Dans les cas où aucune panne ne survient, le couplage avec DSS_OPT permet d'obtenir une solution optimale. La réunion de ces deux algorithmes présente donc certains atouts. En effet, l'ajout d'eFRCD à DSS_OPT ne modifie pas la complexité globale puisqu'elle reste en $0(n^2s^2)$. De plus, on constate dans les faits que le temps consommé uniquement par eFRCD rapporté au temps total utilisé pour exécuter les deux algorithmes est négligeable. En effet, la majorité du temps total d'exécution est occupé par DSS_OPT. L'ajout d'eFRCD à DSS_OPT présente donc à nouveau de sérieux avantages. Pourtant, son implémentation peut probablement être améliorée. En effet, cet algorithme repose sur une heuristique dont le but est de choisir quelles seront les machines qui exécuteront les backups. Dans l'état actuel des choses, cette heuristique se base uniquement sur le temps utilisé pour réaliser les backups sans tenir compte par exemple des temps de communication de la tâche considérée avec son ou ses successeurs, facteur à ne pas négliger surtout si les machines sont éloignées les unes des autres. D'autres heuristiques peuvent être envisagées afin de choisir le plus efficacement possible les serveurs, on pourrait par exemple ne plus sélectionner le meilleur mais le plus mauvais comme on le voit dans l'application du problème des N-reines en Intelligence Artificielle qui donne de bons résultats. Toutefois, la recherche du meilleur serveur présente le gros avantage d'être simple à implémenter à défaut d'être très efficace. De plus, c'est celle qui se rapproche le plus de l'intuition que l'on a naturellement face à un tel problème même si, en ne comparant que les temps d'exécution d'une tâche sur différentes machines sans prendre en compte le temps de communication avec ses voisins, on a une vue partielle du problème. Une autre question peut être soulevée à propos d'eFRCD. En effet, à l'heure actuelle, l'ordonnancement calculé par eFRCD ne résiste qu'à une panne, ainsi comment rendre l'algorithme résistant à un nombre quelconque de pannes ? Il faut que quelque soit la tâche i appartenant au graphe acyclique orienté, $card(\sum_i)$ > nombre de pannes afin de disposer de suffisamment de machines pour pouvoir placer tous les backups de i sur des machines différentes. De plus, il faut placer un nombre de copies de la tâche i égal au nombre de pannes, ainsi on dispose pour i de *nombre de pannes + 1* copies au total.

Enfin, plusieurs observations peuvent également être formulées à propos de l'implémentation même du système que l'on cherche à modéliser. En effet, le graphe ainsi que le stockage des différents coûts est réalisé grâce à des matrices. Or ces dernières, du fait des hypothèses spécifiées plus haut, prennent parfois des formes particulières. En effet, la matrice carrée *lien[][]* utilisée pour représenter le graphe de tâches est triangulaire supérieure. Ainsi, on stocke en mémoire une quantité importante d'éléments alors que seule une partie d'entre eux nous est utile. Dans le cas d'un graphe à 500 tâches, cette matrice nous fournit 250 000 éléments de type entier (soit 4 octets) ce qui représente en mémoire centrale un espace de 1 000 000 octets soit approximativement 1 Mo. Or sur

ces 250 000 éléments disponibles, on ne se sert que des éléments au dessus de la diagonale soit 124 750 éléments qui représentent eux en mémoire un espace de 499 000 octets. On réserve donc plus du double de mémoire que l'on a réellement besoin. Ainsi, bien que l'utilisation des matrices soit pratique, elle n'est pas très efficace et est source d'un vrai gaspillage de ressources. Plusieurs solutions existent afin de résoudre ces problèmes.

Enfin, il reste toujours une question sans réponse : comment évaluer la qualité des ordonnancements ainsi produits ? En effet, aussi bien les résultats fournis par DSS_OPT que par eFRCD ne sont à aucun moment soumis à un critère de qualité ou d'efficacité, hormis le makespan qui nous permet de dire dans quelle mesure notre ordonnancement s'étend dans le temps. Toutefois, plusieurs mesures existent pour mesurer l'efficacité d'un ordonnancement, tel que la notion de *Speed Up*. Malheureusement, un tel critère ne s'applique pas bien ici car il repose sur un environnement de machines homogènes, ce qui n'est pas notre cas. De plus, certaines tâches ne sont pas nécessairement exécutables sur toutes les machines. Enfin, on ne dispose pas ici de contraintes de ressources, on peut donc « superposer » plusieurs exécutions en même temps sur une même machine. Comme on peut le voir, il est difficile d'évaluer la qualité du résultat, car la quantification le temps d'occupation ou d'inoccupation des processeurs n'a pas forcément de sens. Un gros travail reste donc à faire ici afin de vérifier si nos résultats sont vraiment efficaces. De plus, il est possible de faire évoluer ce programme comme nous allons le voir dans la partie suivante.

8. Évolution

Comme nous venons de le voir, l'implémentation actuelle du système pose encore quelques problèmes, d'occupation de mémoire notamment. Ainsi, nous avons vu que l'utilisation des matrices est simple mais a pour principal défaut de nous faire gaspiller de la mémoire. De ce fait, il serait possible d'utiliser d'autres structures plus économiques pour représenter nos données telles que les listes par exemple. En effet, il est tout à fait envisageable de considérer notre graphe de tâche comme une suite d'éléments reliés entre eux par les arcs du graphe. Ainsi, on ne stocke plus de données inutiles. Malheureusement, comme souvent on gagne en efficacité ce que l'on perd en simplicité...En effet, l'implémentation d'une liste est plus compliquée que l'utilisation d'une matrice. Il est également envisageable de concevoir notre système comme l'interaction de plusieurs objets, il suffit alors d'implémenter les classes représentant nos objets tels que les machines, les tâches et les arcs. Enfin, comme nous l'avons vu dans le précédent chapitre, notre implémentation de l'algorithme eFRCD ne supporte qu'une panne. Il serait intéressant de modifier cette méthode de manière à ce qu'elle prenne en entrée un ordonnancement ainsi qu'un nombre de pannes à supporter et qu'en sortie elle retourne l'ordonnancement approprié. A l'heure actuelle, l'implémentation de cet algorithme ne tolère pas que l'on modifie le nombre de pannes à supporter car tout cet aspect est codé en dur. Il faudrait donc revoir quelques points afin de rendre la méthode plus souple à ce niveau. De plus, il serait intéressant de pouvoir fournir à notre méthode eFRCD l'heuristique de notre choix afin d'effectuer des tests. Cela implique donc d'écrire toutes les heuristiques et également de pouvoir passer une d'elles en entrée à la méthode eFRCD.

De manière plus générale, il serait intéressant de modifier l'algorithme eFRCD de manière à gérer non plus des pannes complètes mais partielles, ce qui peut se traduire dans les faits par l'impossibilité pour un serveur d'exécuter un type de tâche par exemple. Il serait également intéressant de prendre en considération des pannes temporaires et non plus permanentes comme c'est le cas actuellement. Un serveur peut dans ce cas tomber en panne pendant un certain temps puis redevenir fonctionnel avant que toutes les tâches aient été exécutées ou inversement. De plus, il peut être utile d'adapter ces deux algorithmes à des problèmes temps-réel comportant des contraintes très fortes

.

9. Conclusion

Finalement, les problèmes d'ordonnancement sont présents partout dans notre vie, il faut donc développer des méthodes afin de répartir le plus efficacement possible les ressources entre les différentes tâches à réaliser. C'est cet objectif qui a conduit aux développement d'algorithmes tels que DSS_OPT et eFRCD. Du fait de leur complémentarité, l'association de ces deux algorithmes fournit une solution à la fois robuste et efficace voire optimale si aucune panne ne survient. Toutefois, bien que puissante par la solution apportée, cette combinaison n'est pas parfaite. En effet, elle est cantonnée aux problèmes relativement petits du fait de la complexité générale en $O(n^2s^2)$. De plus, elle n'est pas applicable sur n'importe quelle machine, notamment celles des particuliers. Ainsi, il serait judicieux de modifier ces algorithmes afin de les adapter à différents types de panne ou bien les rendre utilisables sur un panel plus important de machines, des serveurs multi-processeurs aux machines personnelles. On obtiendrait ainsi une méthode très souple et efficace conservant son optimalité si aucune panne ne survient.

10.Annexes

1. Fichier de trace

```
[MAX_TACHES]4
[MAX_PROCS]3
[Matrice de communication :]
0 1 3
1 0 1
3 1 0

[Matrice de taches :]
29 27 18
14 27 27
19 10 12
23 13 12

[Matrice de liens :]
0 1 9 4
0 0 0 5
0 0 0 7
0 0 0 0
```

2. Fichier de sortie XML

```
<schedule name="trace.xml" makespan="71">
      <resources>
            <resource id="0" name="sigma_2"/>
            <resource id="1" name="sigma_1"/>
            <resource id="2" name="sigma_0"/>
      </resources>
      <tasks>
            <task id="0" name="tache_0" backup="0" start="0" duration="18"
resource="0"/>
            <task id="1" name="tache_0'" backup="1" start="0" duration="27"
resource="1"/>
            <task id="2" name="tache_1" backup="0" start="21" duration="14"
resource="2"/>
            <task id="3" name="tache_1'" backup="1" start="27" duration="27"
resource="1"/>
            <task id="4" name="tache_2" backup="0" start="18" duration="12"
resource="0"/>
            <task id="5" name="tache_2'" backup="1" start="27" duration="10"
resource="1"/>
            <task id="6" name="tache_3" backup="0" start="40" duration="13"
resource="1"/>
            <task id="7" name="tache_3'" backup="1" start="59" duration="12"
resource="0"/>
      </tasks>
</schedule>
```

3. Pseudo-codes

```
/* Pseudo-code de DSS_OPT */
I : ensemble des tâches du graphe
∑ : ensemble des machines disponibles

Début

/* Première partie de l'algorithme */
  Pour chaque i appartenant à I Faire
    Si PRED(i) ≠ vide Alors
      Pour chaque sigma appartenant à ∑i Faire
        b[i][sigma] = 0
        r[i][sigma] = tache[i][sigma]
      Fpour
      Marquer(i)
    FSi
  Fpour
  Tant que (il existe une tache i telle que n'est pas marquée et que tous ses
successeurs le sont) Faire
    Pour chaque sigma appartenant à ∑i Faire
      b[i][sigma] = max k appartenant à PRED(i) (min sigma_p appartenant à
∑k(b[k][sigma_p]+tache[k][sigma_p]+(lien[k][i]*comm[sigma_p][sigma])))
      r[i][sigma] = b[i][sigma] + tache[i][sigma]
    FPour
    Marquer(i)
  FTantQue

  /* Seconde partie de l'algorithme */
  T = max i tel que SUCC(i)=vide (min sigma appartenant à ∑i (r[i][sigma]))
  Pour chaque i tel que SUCC(i) = vide Faire
    min = T
    Pour chaque sigma appartenant ∑i Faire
      Si r[i][sigma] < min Alors
        min = r[i][sigma]
      Fsi
      Récupérer i et sigma tel que r[i][sigma] == min
      Ordonnancer(i,sigma)
    FPour
  FPour
Fin
```

```
/*Pseudo-code de la fonction Ordonnancer */

tache_placee[nombre_tâches][nombre_machines] : matrice booléenne
Entrée : tache_i, sigma_r, nb
Début
  Si tache_placee[tache_i][sigma_r] == Faux Alors
    tache_placee[tache_i][sigma_r] = Vrai
    Si PRED(tache_i) ≠ vide Alors
      Pour chaque k appartenant à PRED(tache_i) Faire
        min = b[tache_i][sigma_r]
        indice_machine = -1
        Pour chaque sigma_p appartenant à ∑k Faire
          Si min >= (r[k][sigma_p]+(lien[k][tache_i]*comm[sigma_p][sigma_r]))
Alors
            min = (r[k][sigma_p]+(lien[k][tache_i]*comm[sigma_p][sigma_r]))
            indice_machine = sigma_p
          FSi
        Fpour
        Ordonnancer(k,indice_machine,nb+1)
      FPour
    FSi
  FSi
Fin
```